Le modèle M.V.C. de Spring

1 Introduction

Le framework Spring [1] est une boite à outils très riche permettant de structurer, d'améliorer et de simplifier l'écriture d'application JEE. Spring est organisé en module

- Gestion des instances de classes (JavaBean et/ou métier),
- Programmation orientée Aspect,
- Modèle MVC et outils pour les application WEB,
- Outils pour la DAO (JDBC),
- Outils pour les ORM (Hibernate, iBatis, JPA, ...),
- Outils pour les applications JEE (EJB, JTA, Servlet, JSP, ...),

Plus d'informations ici [2].

2 Préalables

Durant ce TP nous allons utiliser le conteneur WEB `Tomcat~9.0.x` [3]. Préparez un environement pour tester une application WEB basée sur Spring-MVC :

- Téléchargez (si besoin) `Tomcat~9.0.x` [4] et décompressez l'archive.
- Préparez, dans Eclipse, une application WEB basée sur le conteneur `Tomcat` (utilisez l'adaptateur WTP pour Tomcat 9.0.x).
- Testez une page `index.jsp` très simple.

```
<html>
  <head><title>Example :: Spring Application</title></head>
  <body>
    <h1>Example - Spring Application</h1>
    <p>This is my test.</p>
  </body>
</html>
```

  Vérifiez que cette page est accessible. Faites quelques modifications mineures et testez le résultat.

- Préparez un onglet vers la documentation [5].
- Préparez un onglet vers la Javadoc [6].

3 Mise en place de Spring MVC

3.1 Maven et Spring

- Convertissez votre projet à Maven : **Sélectionnez votre projet** / **Bouton-droit** / **Configure** / **Convert to Maven Project**. Vous devez à cette étape donner une numéro de version à votre projet. Laissez les

1. http://www.springframework.org/
2. https ://docs.spring.io/spring/docs/5.1.x/spring-framework-reference/overview.html
3. http://tomcat.apache.org/
4. http://tomcat.apache.org/
5. https ://docs.spring.io/spring/docs/5.1.x/spring-framework-reference/index.html
6. https ://docs.spring.io/spring/docs/5.1.x/javadoc-api/index.html

valeurs par défaut.

- Ajoutez à votre fichier `pom.xml` Spring MVC :

```
...
<properties>
    <spring.version>5.1.5.RELEASE</spring.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
...
```

- Préparez une classe qui va être le point de démarrage de notre application :

```
package springapp.web;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackageClasses = SpringStart.class)
public class SpringStart implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        System.out.println("Starting....");
        // Init application context
        AnnotationConfigWebApplicationContext webCtx
            = new AnnotationConfigWebApplicationContext();
        webCtx.register(SpringStart.class);
        webCtx.setServletContext(ctx);
        // Init dispatcher servlet
        ServletRegistration.Dynamic servlet
            = ctx.addServlet("springapp", new DispatcherServlet(webCtx));
        servlet.setLoadOnStartup(1);
        servlet.addMapping("*.htm");
        servlet.addMapping("/actions/*");
    }

}
```

Explication : La méthode `onStartup` est automatiquement appelée par Spring pour initialiser l'application. Cette dernière va créer une servlet générique (`DispatcherServlet`) qui va récupérer toutes les requêtes (qui se terminent par `.html` ou qui débutent par `/actions/`) et les aiguiller vers le bon contrôleur. Vous remarquerez que la classe est elle-même une point de configuration Spring (`@Configuration`) qui est enregistré dans le contexte de l'application.

3.2 Un premier contrôleur

- Cette classe Spring va créer un `bean` et lui donner un nom (`/hello.htm`). C'est notre premier contrôleur.

```
package springapp.web;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Service;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

@Service("/hello.htm")
public class HelloController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        logger.info("Returning␣hello␣view");
        return new ModelAndView("/hello.jsp");
    }

}
```

- A cette étape, vous devez trouver dans les traces du serveur la création de ce contrôleur.
- Essayez d'utiliser le contrôleur `hello.htm`
- Visiblement ce contrôleur se termine en donnant la main à une page JSP (`hello.jsp`) destinée à fabriquer la réponse à envoyer au client. Créons cette page :

Page `hello.jsp`

```
<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
  </body>
</html>
```

- Étudiez la classe `ModelAndView` [7]. Cette classe est le coeur du modèle MVC de Spring : Elle permet de séparer d'une part, les contrôleurs qui travaillent sur la requête et d'autres part les vues (pages JSP) qui se chargent du

7. https ://docs.spring.io/spring/docs/5.1.x/javadoc-api/org/springframework/web/servlet/ModelAndView.html

résultat. Entre les deux, les instances de `ModelAndView` transportent à la fois le nom de la vue et les données qui seront affichés par cette vue (c'est-à-dire le **modèle**).

3.3 Améliorer les vues

- Ajoutez à votre fichier `pom.xml` la version 1.2.5 des Spécifications de la JSTL [8].
- Ajoutez à votre fichier `pom.xml` la version 1.2.5 de l'implantation de la JSTL [9].
- Toutes les vues d'une application partagent souvent de nombreuses déclarations. Nous allons les regrouper dans une page JSP. Préparez la page `WEB-INF/jsp/include.jsp` suivante :

```
<%@ page session="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Nous pouvons maintenant revoir notre page d'accueil (fichier `index.jsp`) en forçant l'utilisateur à utiliser le contrôleur :

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<%-- rediriger le contrôleur hello --%>
<c:redirect url="/hello.htm"/>
```

Testez le bon fonctionnement de cette redirection.

- Pour l'instant la page `hello.jsp` est accessible au client dans la mesure ou le client connait le nom de cette page. Dans le modèle MVC, les requêtes doivent **toujours** être traitées par le contrôleur et non pas, directement, par une page JSP. Pour éviter que la page `hello.jsp` ne soit accessible, nous allons la modifier et la déplacer dans `WEB-INF/jsp/hello.jsp` :

```
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
  <head><title>Hello :: Spring Application</title></head>
  <body>
    <h1>Hello - Spring Application</h1>
    <p>Greetings, it is now <c:out value="${now}" default="None" /></p>
  </body>
</html>
```

- Bien entendu, nous devons modifier le contrôleur en conséquence :

8. https ://mvnrepository.com/artifact/org.apache.taglibs/taglibs-standard-spec
9. https ://mvnrepository.com/artifact/org.apache.taglibs/taglibs-standard-impl

```java
package springapp.web;

import java.io.IOException;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Service;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

@Service("/hello.htm")
public class HelloController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    public ModelAndView handleRequest(HttpServletRequest request, //
            HttpServletResponse response) throws ServletException, IOException {

        String now = (new Date()).toString();
        logger.info("Returning␣hello␣view␣with␣" + now);

        return new ModelAndView("WEB-INF/jsp/hello.jsp", "now", now);
    }

}
```

Nous avons profité de cette mise à jour pour introduire **une nouveauté** : le contrôleur fabrique maintenant une donnée et transmet cette donnée à la page JSP qui se charge de l'afficher (la date).

Exercice : préparez une nouvelle donnée (par exemple un message), passez cette donnée à la page `hello.jsp` et assurez sa présentation.

3.4 Découpler les vues et les contrôleurs

Pour l'instant, le chemin d'accès complet à la vue figure dans le contrôleur. Pour éviter ce couplage fort, nous allons créez un service **spring** qui va se charger de retrouver les vues à partir d'un simple nom. Ajoutez à la classe `SpringStart` le code :

```java
@Bean
public ViewResolver viewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setViewClass(JstlView.class);
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

doté de ce service de résolution des noms de vue, le code du contrôleur va devenir :

```java
package springapp.web;

import java.io.IOException;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Service;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

@Service("/hello.htm")
public class HelloController implements Controller {

    protected final Log logger = LogFactory.getLog(getClass());

    public ModelAndView handleRequest(HttpServletRequest request, //
            HttpServletResponse response) throws ServletException, IOException {

        String now = (new Date()).toString();
        logger.info("Returning␣hello␣view␣with␣" + now);

        return new ModelAndView("hello", "now", now);
    }

}
```

Exercice : A ce stade, vous pouvez faire un classe de test unitaire pour vérifier que votre contrôleur est bien correct.

4 Utiliser les annotations

- Nous pouvons maintenant définir un nouveau contrôleur :

```java
package springapp.web;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller()
@RequestMapping("/tests")
public class HelloAnnoController {

    protected final Log logger = LogFactory.getLog(getClass());

    @RequestMapping(value = "/welcome", method = RequestMethod.GET)
    public ModelAndView sayHello() {
        String now = (new Date()).toString();
        logger.info("Running␣" + this);
        return new ModelAndView("hello", "now", now);
    }

}
```

Exercices :

- Vérifiez dans les traces du serveur que ces contrôleurs sont bien détectés par Spring.
- Testez ce contrôleur avec une URL du type

  `http://localhost:8080/votre-application/actions/tests/welcome`

- Lisez la documention de l'annotation `@Controller`.
- Un contrôleur est maintenant une méthode qui
  - ▷ renvoie une instance de `ModelAndView` ou le nom d'une vue (`String`),
  - ▷ accepte en argument une instance de `HttpServletRequest` et/ou `HttpServletRequest` et/ou `HttpSession` et bien d'autres choses.
- Créez un nouveau contrôleur (`/actions/tests/counter`) qui va stocker un compteur en session, le faire évoluer et assurer son affichage. Faites en sorte que ce contrôleur traite également un argument de la requête HTTP.

4.1 Déclarer les paramètres

Nous pouvons également utiliser l'annotation `@RequestParam` pour récupérer, sous la forme d'un paramètre de la méthode, les paramètres de la requête HTTP. En voici un exemple :

```java
@RequestMapping(value = "/plus10", method = RequestMethod.GET)
public ModelAndView plus10(
        @RequestParam(value = "num", defaultValue = "100") Integer value) {
    logger.info("Running␣plus10␣controler␣with␣param␣=␣" + value);
    return new ModelAndView("hello", "now", value + 10);
}
```

Exercices :

- Testez ce contrôleur en lui fournissant le paramètre attendu. Testez également les cas d'erreur (paramètre absent ou incorrect).
- Ajoutez un nouveau paramètre de type `Date` et utilisez l'annotation `@DateTimeFormat` pour récupérer ce paramètre.

4.2 Utiliser de belles adresses

Il est maintenant habituel de placer des paramètres à l'intérieur des adresses WEB. cela permet d'avoir des URL simples, faciles à construire et faciles à mémoriser. En voila un exemple :

```
@RequestMapping(value = "/voir/{param}", method = RequestMethod.GET)
public ModelAndView voir(@PathVariable("param") Integer param) {
    logger.info("Running␣param␣controler␣with␣param=" + param);
    return new ModelAndView("hello", "now", param);
}
```

Exercices :

- Testez ce contrôleur.
- Modifiez ce contrôleur pour avoir plusieurs paramètres dans la même adresse.
- Utilisez le mécanisme des expressions régulières pour traiter une adresse composée (inspirez de la documention sur l'annotation `@PathVariable`).
- Terminez en testant l'annotation `@MatrixVariable` pour traiter des URL de la forme (`a` et `b` ne sont pas des paramètres) :

```
/une/action;a=10;b=20
```

5 Le traitement des formulaires

Pour introduire la traitement des formulaires nous allons procéder en trois étapes :

- définition d'un modèle et d'une couche métier,
- création du formulaire,
- validation des données,

5.1 Définir une couche métier

Prévoir une configuration Spring :

- Créez la paquetage `springapp.model`
- Créez la paquetage `springapp.business`
- Prévoir une classe de configuration de la couche métier (vide pour l'instant) :

```java
package springapp.business;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackageClasses = SpringBusinessConfig.class)
public class SpringBusinessConfig {

}
```

- Ajoutez à la classe `SpringStart` une annotation d'importation pour que la configuration de la couche métier soit prise en compte :

```java
@Import(SpringBusinessConfig.class)
```

Remarque : Il est important de segmenter la configuration Spring en plusieurs blocs. Vous pourrez ainsi prévoir des tests unitaires de la couche métier en utilisant la configuration adéquate sans être encombré par d'autres composants.

Considérons le POJO (*Plain Old java Object*) suivant :

```java
package springapp.model;

public class Product {

    private Integer number;
    private String name;
    private Double price;
    private String description;
    private String type;

    public Integer getNumber() {
        return number;
    }

    public void setNumber(Integer number) {
        this.number = number;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

}
```

Nous allons lui adjoindre un service métier défini par l'interface ci-dessous :

```java
package springapp.business;

import java.util.Collection;

import springapp.model.Product;

public interface IProductManager {

    Collection<Product> findAll();

    void save(Product p);

    Product find(int number);

}
```

pour laquelle nous définissons une première implantation :

```java
package springapp.business;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import org.springframework.stereotype.Service;

import springapp.model.Product;

@Service("productManager")
public class InMemoryProductManager implements IProductManager {

    final Map<Integer, Product> products;
    int maxId = 0;

    public InMemoryProductManager() {
        this.products = new HashMap<Integer, Product>();
        Product p1 = new Product();
        p1.setNumber(100);
        p1.setName("Car");
        p1.setPrice(2000.0);
        p1.setDescription("Small␣car");
        products.put(p1.getNumber(), p1);
        Product p2 = new Product();
        p2.setNumber(200);
        p2.setName("Gift");
        p2.setPrice(100.0);
        p2.setDescription("Big␣gift");
        products.put(p2.getNumber(), p2);
        maxId = 300;
    }

    @Override
    public Collection<Product> findAll() {
        return products.values();
    }

    @Override
    public void save(Product p) {
        if (p.getNumber() == null) {
            p.setNumber(maxId++);
        }
        products.put(p.getNumber(), p);
    }

    @Override
    public Product find(int number) {
        Product p = products.get(number);
        if (p == null) {
            throw new IllegalArgumentException("no␣product␣" + number);
        }
        return p;
    }
}
```

5.2 Lister les produits

Nous pouvons maintenant mettre en place un contrôleur qui va gérer toutes les actions sur les produits (listage, création, modification et suppression).

Commençons par lister les produits disponibles :

```
package springapp.web;

import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.Map;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import springapp.business.IProductManager;
import springapp.model.Product;

@Controller()
@RequestMapping("/product")
public class ProductController {

    @Autowired
    IProductManager manager;

    protected final Log logger = LogFactory.getLog(getClass());

    @RequestMapping(value = "/list", method = RequestMethod.GET)
    public ModelAndView listProducts() {
        logger.info("List␣of␣products");
        Collection<Product> products = manager.findAll();
        return new ModelAndView("productsList", "products", products);
    }

}
```

Ce contrôleur est accompagné de la vue `productsList.jsp` :

```
<%@ include file="/WEB-INF/jsp/include.jsp"%>

<c:url var="edit" value="/actions/product/edit" />

<html>
<head>
<title>Hello :: Spring Application</title>
</head>
<body>
    <h1>Products</h1>
    <table border='1'>
    <c:forEach items="${products}" var="prod">
        <tr>
        <td><a href="${edit}?id=${prod.number}">
            <c:out value="${prod.name}" /></a>
        </td>
        <td><i>$<c:out value="${prod.price}" /></i></td>
        </tr>
    </c:forEach>
    </table>
    <p><a href="${edit}">Create new product</a></p>
</body>
</html>
```

Cette vue va construire la liste des produits, avec pour chacun une possibilité d'édition. Bien entendu, pour l'instant, la phase d'édition ne fonctionne pas.

Note : Dans cet exemple, le contrôleur ne fait rien d'autre que de construire des données (la liste des produits) pour les envoyer à la vue. La création des données peut être découplée des contrôleurs et placée dans des méthodes annotées par `@ModelAttribute`. Ces méthodes sont systématiquement exécutées avant les contrôleurs pour remplir le modèle.

Dans notre exemple, la création de la liste des produits (nommée `products`) peut se faire par la méthode :

```
@ModelAttribute("products")
Collection<Product> products() {
    logger.info("Making␣list␣of␣products");
    return manager.findAll();
}
```

Le contrôleur devient :

```
@RequestMapping(value = "/list", method = RequestMethod.GET)
public String listProducts() {
    logger.info("List␣of␣products");
    return "productsList";
}
```

La vue va simplement puiser dans le modèle qui est rempli par la méthode `products`.

5.3 Utiliser bootstrap

Pour améliorer l'aspect de nos pages, nous allons utiliser le framework Bootstrap [10]. Pour ce faire

- Créez la page `head-bootstrap.jsp` ci-dessous qui regroupe la déclaration des ressources CSS et JavaScript nécessaires :

10. http ://getbootstrap.com/

```
<link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js">
</script>
<script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
</script>
```

- Modifiez votre page `productsList.jsp` pour utiliser la structure et les classes offertes par bootstrap :

```
<!DOCTYPE html>
<%@ include file="/WEB-INF/jsp/include.jsp"%>

<c:url var="edit" value="/actions/product/edit" />

<html>
<head>
<title>Hello :: Spring Application</title>
<%@ include file="/WEB-INF/jsp/head-bootstrap.jsp"%>
</head>
<body>
    <div class="container">
        <h1>Products (bootstrap)</h1>
        <table class="table␣table-hover">
            <c:forEach items="${products}" var="prod">
                <tr>
                    <td><a href="${edit}?id=${prod.number}">
                        <c:out value="${prod.name}" />
                    </a></td>
                    <td><i>$<c:out value="${prod.price}" /></i></td>
                </tr>
            </c:forEach>
        </table>
        <p>
            <a class="btn␣btn-info" href="${edit}">Create new product</a>
        </p>
    </div>
</body>
</html>
```

- Utilisez cette petite documentation [11] pour découvrir les possibilités de bootstrap.

5.4 Éditer un produit

Définissons maintenant le contrôleur d'accès au formulaire d'édition :

```
@RequestMapping(value = "/edit", method = RequestMethod.GET)
public String editProduct(@ModelAttribute Product p) {
    return "productForm";
}
```

accompagné du formulaire `productForm.jsp` :

11. https://www.w3schools.com/bootstrap/default.asp

```
<!DOCTYPE html>

<%@ include file="/WEB-INF/jsp/include.jsp"%>

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<html>
<head>
<%@ include file="/WEB-INF/jsp/head-bootstrap.jsp"%>
</head>

<body>

    <div class="container">
        <h1>Edit Product</h1>

        <form:form method="POST" modelAttribute="product">

            <form:errors path="*" cssClass="alert␣alert-danger" element="div" />

            <div class="form-group">
                <label for="name">Name:</label>
                <form:input class="form-control" path="name" />
                <form:errors path="name" cssClass="alert␣alert-warning"
                    element="div" />
            </div>
            <div class="form-group">
                <label for="description">Description:</label>
                <form:textarea class="form-control" path="description" rows="4" />
                <form:errors path="description" cssClass="alert␣alert-warning"
                    element="div" />
            </div>
            <div class="form-group">
                <label for="price">Price:</label>
                <form:input path="price" class="form-control" />
                <form:errors path="price" cssClass="alert␣alert-warning"
                    element="div" />
            </div>
            <div class="form-group">
                <button type="submit" class="btn␣btn-info">Submit</button>
            </div>
        </form:form>
    </div>

</body>
</html>
```

Cette vue utilise les balises personnalisées de Spring pour gérer facilement la récupération des données du modèle (attribut `modelAttribute` de la balise `form`) et la mise en place des champs (balises `form:input`, `form:select`, etc...). Vous trouverez plus d'information sur ces balises dans cette documentation [12].

Pour l'instant, ce formulaire ne permet pas d'éditer des produits déjà existants. Pour ce faire, nous allons ajouter une méthode annotée `@ModelAttribute` qui va préparer l'instance du produit à éditer en fonction du paramètre de la requête HTTP :

12. http://docs.spring.io/spring/docs/4.3.2.RELEASE/spring-framework-reference/html/view.html#view-jsp-formtaglib

```
@ModelAttribute
public Product newProduct(
        @RequestParam(value = "id", required = false) Integer productNumber) {
    if (productNumber != null) {
        logger.info("find␣product␣" + productNumber);
        return manager.find(productNumber);
    }
    Product p = new Product();
    p.setNumber(null);
    p.setName("");
    p.setPrice(0.0);
    p.setDescription("");
    logger.info("new␣product␣=␣" + p);
    return p;
}
```

En clair : si la requête `/edit` est accompagnée d'un numéro de produit (paramètre `id` optionnel), le produit sera chargé à partir du manager. Dans le cas contraire, un nouveau produit sera renvoyé. Testez ce fonctionnement.

Il nous reste maintenant à mettre en place le contrôleur de soumission du formulaire :

```
@RequestMapping(value = "/edit", method = RequestMethod.POST)
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    if (result.hasErrors()) {
        return "productForm";
    }
    manager.save(p);
    return "productsList";
}
```

A cette étape, la seule erreur possible provient d'une erreur de conversion sur le prix. Essayez de donner un prix incorrect afin de tester ce fonctionnement.

Avertissement : A ce stade, la création d'un nouveau produit (après la soumission) se termine sur l'affichage de la liste des produits (dernière ligne du contrôleur ci-dessus). Ce comportement pose un problème : Si le client tente un rechargement de la page, cela va provoquer une nouvelle soumission et la création d'un nouveau produit !

Pour régler ce problème, nous allons renvoyer non pas sur la vue `productsList`, mais sur l'action permettant d'avoir la liste des produits :

```
@RequestMapping(value = "/edit", method = RequestMethod.POST)
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    if (result.hasErrors()) {
        return "productForm";
    }
    manager.save(p);
    return "redirect:list";
}
```

5.5 Injecter des données

Pour construire un formulaire complexe, nous avons souvent besoin d'utiliser des données annexes (liste de références, nom d'utilisateur, etc.). Pour ce faire, nous allons de nouveau utiliser l'annotation `@ModelAttribute`. Mettez en place la méthode suivante :

```
@ModelAttribute("productTypes")
public Map<String, String> productTypes() {
    Map<String, String> types = new LinkedHashMap<>();
    types.put("type1", "Type␣1");
    types.put("type2", "Type␣2");
    types.put("type3", "Type␣3");
    types.put("type4", "Type␣4");
    types.put("type5", "Type␣5");
    return types;
}
```

Elle fabrique et injecte dans le modèle une table de correspondance qui va nous être utile pour ajouter le champ de typage dans le formulaire. Modifiez notre formulaire en ajoutant :

```
<div class="form-group">
    <label for="type">Type:</label>
    <form:select path="type" multiple="false" class="form-control">
        <form:option value="" label="---␣Select␣---" />
        <form:options items="${productTypes}" />
    </form:select>
    <form:errors path="type" cssClass="alert␣alert-warning"
        element="div" />
</div>
```

Nous pouvons maintenant associer un type à chaque produit.

5.6 Valider les données

Il manque maintenant la phase de validation des données du formulaire. Pour ce faire, nous allons développer une classe de validation adaptée au produit :

```java
package springapp.web;

import org.springframework.stereotype.Service;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import springapp.model.Product;
import springapp.model.ProductCode;

@Service
public class ProductValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
                "product.name", "Field␣name␣is␣required.");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "description",
                "product.description", "Field␣description␣is␣required.");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "type",
                "product.type", "Field␣type␣is␣required.");

        if (!(product.getPrice() > 0.0)) {
            errors.rejectValue("price", "product.price.too.low",
                    "Price␣too␣low");
        }
    }

}
```

Pour l'utiliser, il suffit de modifier le contrôleur comme suit :

```java
@Autowired
ProductValidator validator;

@RequestMapping(value = "/edit", method = RequestMethod.POST)
public String saveProduct(@ModelAttribute Product p, BindingResult result) {
    validator.validate(p, result);
    if (result.hasErrors()) {
        return "productForm";
    }
    manager.save(p);
    return "productsList";
}
```

5.7 Traduire les messages de validation

Pour l'instant les messages d'erreurs sont affichés en anglais. Nous pouvons les traduire automatiquement en délocalisant ces messages dans des fichiers de ressources.

Commencez par créer dans le répertoire contenant les sources du paquetage `springapp.web` le fichier `product.properties` :

```
product.name = Name is required!
product.description = Description is required!
product.type = Type is required!
product.price.too.low = Price is too low!
```

puis le fichier `product_fr_FR.properties` :

```
product.name = Le nom est requis
product.price.too.low = Le prix est trop bas !
```

Tous les messages sont donnés en anglais. Certains sont en français.

Pour exploiter ces ressources, nous allons les charger en ajoutant dans la classe `SpringStart` la création d'un nouveau service :

```
@Bean("messageSource")
public ResourceBundleMessageSource messageSource() {
    ResourceBundleMessageSource r = new ResourceBundleMessageSource();
    r.setBasenames("/springapp/web/product");
    return r;
}
```

Nous pouvons simplifier notre classe de validation en supprimant les messages. Elle devient :

```java
package springapp.web;

import org.springframework.stereotype.Service;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import springapp.model.Product;
import springapp.model.ProductCode;

@Service
public class ProductValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
                "product.name");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "description",
                "product.description");

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "type",
                "product.type");

        if (!(product.getPrice() > 0.0)) {
            errors.rejectValue("price", "product.price.too.low");
        }
    }

}
```

À faire : Suivez ce tutoriel pour changer la langue sur la demande de l'utilisateur [13].

5.8 Traiter des champs complexes

Nous venons de le voir, Spring MVC traite parfaitement les champs qui correspondent à un type de base (entier, flottant, chaîne, etc.). Nous allons maintenant nous occuper des champs complexes.

Commençons par définir une classe pour représenter le numéro de série d'un produit (une lettre suivie d'un entier entre 1000 et 9999) :

13. https://www.mkyong.com/spring-mvc/spring-mvc-internationalization-example/

```java
package springapp.model;

public class ProductCode {

    String base;
    int number;

    public String getBase() {
        return base;
    }

    public void setBase(String base) {
        this.base = base;
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public ProductCode() {
        super();
    }

    public ProductCode(String base, int number) {
        super();
        this.base = base;
        this.number = number;
    }

}
```

et ajoutons ce code à notre produit :

```java
package springapp.model;

public class Product {

    ...

    private ProductCode code;

    ...

    public ProductCode getCode() {
        return code;
    }

    public void setCode(ProductCode code) {
        this.code = code;
    }

}
```

Faites ensuite les modifications suivantes :

- dans `InMemoryProductManager` associez le code `A1000` au premier produit et `B2000` au deuxième.
- ajoutez le code suivant à votre formulaire :

```
<div class="form-group">
    <label for="code">Code:</label>
    <form:input path="code.base" class="form-control"/>
    <form:input path="code.number" class="form-control"/>
    <form:errors path="code" cssClass="alert␣alert-warning"
        element="div" />
</div>
```

- Modifiez le validateur en conséquence en ajoutant

```
ProductCode code = product.getCode();
if (code != null) {
    if (!code.getBase().matches("[A-Z]")) {
        errors.rejectValue("code", "product.code.base");
    }
    if (!(code.getNumber() >= 1000 && code.getNumber() <= 9999)) {
        errors.rejectValue("code", "product.code.number");
    }
}
```

- Ajoutez des messages d'erreurs pour `product.code.base` et `product.code.number`.

Vous devez maintenant être capable d'éditer les deux parties du numéro de série.

Une autre solution consiste à fournir une classe d'adaptation (un éditeur dans la terminologie des JavaBeans) qui est capable de transformer un code en chaîne et vice-versa. En voici un exemple :

```
package springapp.web;

import java.beans.PropertyEditorSupport;

import springapp.model.ProductCode;

class ProductCodeEditor extends PropertyEditorSupport {

    @Override
    public String getAsText() {
        Object o = this.getValue();
        if (o instanceof ProductCode) {
            ProductCode c = (ProductCode) o;
            return c.getBase() + "" + c.getNumber();
        }
        return super.getAsText();
    }

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        try {
            String base = text.substring(0, 1);
            int number = Integer.parseInt(text.substring(1));
            ProductCode c = new ProductCode(base, number);
            super.setValue(c);
        } catch (Exception e) {
            throw new IllegalArgumentException("Bad␣code␣format");
        }
    }

}
```

Il suffit maintenant d'indiquer au contrôleur que nous disposons de cette classe. Pour ce faire nous allons lui adjoindre une méthode annotée par `InitBinder` :

```
@InitBinder
public void initBinder(WebDataBinder b) {
    b.registerCustomEditor(ProductCode.class, new ProductCodeEditor());
}
```

Le formulaire peut devenir :

```
<div class="form-group">
    <label for="code">Code:</label>
    <form:input path="code" class="form-control" />
    <form:errors path="code" cssClass="alert␣alert-warning"
        element="div" />
</div>
```

Testez le bon fonctionnement de cette nouvelle version.

6 Validation des JavaBean dans JEE 6/7/8

Une nouvelle spécification (JSR303) nous permet d'exprimer les contraintes sur les propriétés par des annotations (plus d'information dans la documentation JEE 6 [14] et dans la JavaDoc [15]).

6.1 Mise en oeuvre

Commençons par ajouter la dépendance pour l'implantation d'Hibernate [16] de la spécification JSR 303.

Continuons en utilisant les annotations dans nos POJOs :

14. http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html
15. http ://docs.oracle.com/javaee/6/api/javax/validation/constraints/package-summary.html
16. https ://mvnrepository.com/artifact/org.hibernate/hibernate-validator

```
package springapp.model;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Product {

    private Integer number;

    @NotNull
    @Size(min = 1, message = "Le nom est obligatoire")
    private String name;

    @NotNull
    @Min(value = 1, message = "Le prix est trop bas")
    private Double price;

    @NotNull(message = "La description est obligatoire")
    @Size(min = 1, max = 100, message = "Entre 1 et 200 caractères")
    private String description;

    @NotNull()
    @Size(min=1,message="Le type doit être renseigné")
    private String type;

    @Valid
    private ProductCode code;

    ... getters and setters

}
```

et

```
package springapp.model;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class ProductCode {

    @NotNull
    @Size(min = 1, max = 1)
    @Pattern(regexp="[A-Z]", message="Le code doit débuter par une majuscule")
    String base;

    @Min(value=1000, message="Le numéro doit être >= à 1000")
    @Max(value=9999, message="Le numéro doit être <= à 9999")
    int number;

    ... getters and setters

}
```

Nous allons ajouter l'annotation `@Valid` dans le contrôleur :

```
...

@RequestMapping(value = "/edit", method = RequestMethod.POST)
public String saveProduct(@ModelAttribute @Valid Product p, BindingResult result) {
    validator.validate(p, result);
    if (result.hasErrors()) {
        return "productForm";
    }
    manager.save(p);
    return "productsList";
}

...
```

Testez le résultat. Vous devez vous retrouver avec les messages en provenance du validateur plus les nouveaux messages en provenance des annotations. Vous pouvez maintenant vider votre validateur manuel. La classe de validation reste **utile** pour les **validations métier**.

6.2 Créer ses propres contraintes

Le mécanisme de validation peut facilement être étendu pas ajout de contraintes spécifiques. Nous allons créer une contrainte `Bye` qui va vérifier la présence de ce mot dans un champ. Pour ce faire, commencez par créez l'annotation `Bye` :

```
package springapp.web;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ByeConstraintValidator.class)
@Documented
public @interface Bye {

    String message() default "Il␣manque␣le␣'bye'";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Puis continuez en créant la classe de validation `ByeConstraintValidator` :

```java
package springapp.web;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class ByeConstraintValidator implements ConstraintValidator<Bye, String> {

    @Override
    public void initialize(Bye arg0) {
    }

    @Override
    public boolean isValid(String arg0, ConstraintValidatorContext arg1) {
        if (arg0.contains("bye"))
            return true;
        return false;
    }

}
```

Modifez ensuite la classe `Product` pour utiliser cette annotation :

```java
public class Product {

    ...

    @NotNull(message = "La description est obligatoire")
    @Size(min = 1, max = 100, message = "Entre 1 et 200 caractères")
    @Bye
    private String description;

    ...

}
```

Vérifiez son bon fonctionnement !

7 Utiliser des données en session

Il est facile de récupérer des données placées en session, mais Spring nous offre le moyen d'injecter directement dans nos contrôleurs des données de portée session.

Étape 1 : définissez un nouveau bean pour représenter l'utilisateur courant :

```java
package springapp.web;

import java.io.Serializable;

import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component()
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

L'annotation `Component` indique que c'est un composant géré par Spring. L'annotation `Scope` donne la portée des instances (une par session). La clause `ProxyMode` permet d'indiquer que ce n'est pas directement une instance qui doit être injectée, mais un proxy qui va sélectionner la bonne instance (dans la bonne session) en fonction du contexte.

Étape 2 : définissez un contrôleur qui utilise l'injection du bean `User` :

```java
package springapp.web;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller()
@RequestMapping("/user")
public class UserController {

    protected final Log logger = LogFactory.getLog(getClass());

    @Autowired()
    User user;

    @ModelAttribute("user")
    public User newUser() {
        return user;
    }

    @RequestMapping(value = "/show")
    public String show() {
        logger.info("show␣user␣" + user);
        return "user";
    }

    @RequestMapping(value = "/login")
    public String login() {
        logger.info("login␣user␣" + user);
        user.setName("It's␣me");
        return "user";
    }

    @RequestMapping(value = "/logout")
    public String logout() {
        logger.info("logout␣user␣" + user);
        user.setName("Anonymous");
        return "user";
    }
}
```

Étape 3 : La vue :

```
<%@ include file="/WEB-INF/jsp/include.jsp"%>

<c:url var="login" value="/actions/user/login" />
<c:url var="logout" value="/actions/user/logout" />
<c:url var="show" value="/actions/user/show" />

<html>
<body>
    <h1>User</h1>

    <p>
    name : <c:out value="${user.name}" default="no␣name"/> |
    <a href="${show}">Show</a> | <a href="${login}">Login</a> |
    <a href="${logout}">Logout</a>
    </p>
</body>
</html>
```

Moralité : Le contrôleur (qui est un singleton exécuté par plusieurs `threads`) utilise le `proxy` pour sélectionner **automatiquement** l'instance du bean `User` qui correspond à la requête courante et à la session courante.

La liaison se fait par le `thread`. C'est le même `thread` qui traite toute la requête (`Dispatcher`, contrôleur, vue). Le `thread` courant est donc utilisé comme une sorte de variable globale qui permet de faire des liaisons implicites.

8 Utiliser JPA

Nous allons maintenant étudier comment mettre en place une couche métier basée sur `JPA` (Java Persistence API)

8.1 Préparer les outils

Ajoutez à votre fichier `pom.xml` la version d'Hibernate (en plus de celle de Spring) :

```
...
<properties>
    <spring.version>5.1.5.RELEASE</spring.version>
    <hibernate.version>5.2.3.Final</hibernate.version>
</properties>
...
```

puis les dépendances nécessaires :

```
...
<!-- ============================================== -->
<!-- === LA PARTIE DONNÉES ======================== -->
<!-- ============================================== -->
<!-- pour utiliser JPA avec Spring -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- pour utiliser Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<!-- nécessaire pour Hibernate avec java > 8 -->
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>2.4.0-b180830.0438</version>
</dependency>
<!-- la BD HSQL -->
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.4.0</version>
</dependency>
...
```

8.2 Définition des entités

Prenons un JavaBean pour représenter un message :

```java
package springapp.business;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@NamedQueries({
    @NamedQuery(name = "Message.findAll", query = "Select m From Message m"),
    @NamedQuery(name = "Message.removeAll", query = "Delete From Message")
})

@Entity
public class Message {

    @Id
    @GeneratedValue
    Integer number;

    @Column
    String text;

    public Message() {
        super();
    }

    public Message(String text) {
        super();
        this.text = text;
    }

    public Integer getNumber() {
        return number;
    }

    public void setNumber(Integer number) {
        this.number = number;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    @Override
    public String toString() {
        return String.format("Message [number=%s, text=%s]", number, text);
    }

}
```

et un service Spring en commençant par la spécification :

```java
package springapp.business;

import java.util.Collection;

public interface IMessageManager {

    void add(String message);

    int removeAll();

    Collection<Message> findAll();

}
```

puis en continuant par l'implantation

```java
package springapp.business;

import java.util.Collection;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
@Transactional
public class MessageManager implements IMessageManager {

    @Autowired
    @PersistenceContext(unitName = "myData")
    EntityManager em;

    @Override
    public void add(String data) {
        Message m = new Message(data);
        em.persist(m);
        System.out.println("NEW␣MESSAGE␣=␣" + m);
    }

    @Override
    public int removeAll() {
        return em.createNamedQuery("Message.removeAll").executeUpdate();
    }

    @Override
    public Collection<Message> findAll() {
        return em.createNamedQuery("Message.findAll", Message.class).getResultList();
    }
}
```

Vous pouvez noter que nous utilisons l'injection de dépendances pour demander à Spring de préparer et d'injecter une instance d'un `EntityManager`.

Créez dans le répertoire `src` le fichier de propriétés `config.properties` :

```
datasource.driverName=org.hsqldb.jdbcDriver
datasource.user=SA
datasource.password=
datasource.url=jdbc:hsqldb:file:data/mydb
# Si problème : jdbc:hsqldb:file:data/mydb;shutdown=true
```

Enrichissons La classe de configuration de notre couche métier (`SpringBusinessConfig`) en prévoyant une source de données, une usine à EM, et un gestionnaire de transaction :

```java
package springapp.business;

import java.util.Properties;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@ComponentScan(basePackageClasses = SpringBusinessConfig.class)
@EnableTransactionManagement
@PropertySource("classpath:config.properties")
public class SpringBusinessConfig {

    /*
     * Définition de la source de données
     */
    @Bean
    public DataSource dataSource(//
            @Value("${datasource.driverName}") String driverName, //
            @Value("${datasource.url}") String url, //
            @Value("${datasource.user}") String user, //
            @Value("${datasource.password}") String password) {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(driverName);
        dataSource.setUrl(url);
        dataSource.setUsername(user);
        dataSource.setPassword(password);
        return dataSource;
    }

    /*
     * Construction de l'entityManagerFactory à partir de la source de données et du
     * choix d'hibernate. Cette configuration remplace le fichier persistence.xml
     */
    @Bean(name = "myData")
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(//
            @Autowired DataSource ds) {
        LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();
        em.setDataSource(ds);
        em.setPackagesToScan(new String[] { "springapp.business" });
        em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        em.setBeanName("myData");
        // Configuration d'hibernate
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
        properties.setProperty("hibernate.show_sql", "true");
        properties.setProperty("hibernate.format_sql", "true");
        em.setJpaProperties(properties);
        return em;
    }
```

8.3 La partie WEB

Commençons par le contrôleur :

```
package springapp.web;

import java.util.Collection;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

import springapp.business.IMessageManager;
import springapp.business.Message;

@Controller()
@RequestMapping("/message")
public class MessageController {

    @Autowired
    IMessageManager messageManger;

    protected final Log logger = LogFactory.getLog(getClass());

    @RequestMapping(value = "/add")
    public ModelAndView add(@RequestParam(required = true) String text) {
        messageManger.add(text);
        return new ModelAndView("message", "messages", messages());
    }

    @RequestMapping(value = "/removeAll")
    public ModelAndView removeAll() {
        int n = messageManger.removeAll();
        logger.info(n + " deleted message(s)");
        return new ModelAndView("message", "messages", messages());
    }

    @RequestMapping(value = "/list")
    public String list() {
        return "message";
    }

    @ModelAttribute("messages")
    public Collection<Message> messages() {
        return messageManger.findAll();
    }
}
```

et terminons par la vue :

```
<!DOCTYPE html>

<%@ include file="/WEB-INF/jsp/include.jsp"%>

<c:url var="add" value="/actions/message/add" />
<c:url var="remove" value="/actions/message/removeAll" />
<c:url var="list" value="/actions/message/list" />

<html>
<head>
<%@ include file="/WEB-INF/jsp/head-bootstrap.jsp"%>
</head>
<body>
    <div class="container">
        <h1>Messages</h1>

        <form action="${add}" method="POST" class="form-inline">
            <div class="form-group">
                <input name="text" size="20" class="form-control" />
            </div>
            <div class="form-group">
                <input type="submit" value="Add" class="form-control btn btn-info" />
            </div>
            <div class="form-group">
                <a class="btn btn-success" href="${list}">List</a>
            </div>
            <div class="form-group">
                <a class="btn btn-danger" href="${remove}">Remove All</a>
            </div>
        </form>

        <table class="table table-hover">
            <c:forEach items="${messages}" var="m">
                <tr>
                    <td><c:out value="${m.number}" /></td>
                    <td><c:out value="${m.text}" /></td>
                </tr>
            </c:forEach>
        </table>
    </div>
</body>
</html>
```

Testez le bon fonctionement de votre application.

9 Enrichir notre application

Pour revoir la totalité du processus, ajoutez à votre application une fonction de suppression. Vous devrez

- revoir la définition du service métier et son implantation,
- ajouter une action de suppression,
- offrir cette action dans la vue de listage.

10 C'est fini

À bientôt.